Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-011-0186-x

REGULAR PAPER

Modelling and analysis using GROOVE

Amir Hossein Ghamarian · Maarten de Mol · Arend Rensink · Eduardo Zambon · Maria Zimakova

Abstract In this paper we present case studies that describe how the graph transformation tool GROOVE has been used to model problems from a wide variety of domains. These case studies highlight the wide applicability of GROOVE in particular, and of graph transformation in general. They also give concrete templates for using GROOVE in practice. Furthermore, we use the case studies to analyse the main strong and weak points of GROOVE.

Keywords Graph-based modelling · Graph transformation · Tool application · Model transformation · State space exploration

1 Introduction

In this paper, we take the perspective of one particular modelling paradigm, *graph transformation*, and one particular tool supporting this paradigm, GROOVE. We target the following groups of readers:

A. H. Ghamarian · M. de Mol (✉) · A. Rensink · E. Zambon · M. Zimakova
Department of Computer Science, University of Twente, Enschede, The Netherlands
e-mail: molm@cs.utwente.nl

A. H. Ghamarian
e-mail: ghamarian@cs.utwente.nl

A. Rensink
e-mail: rensink@cs.utwente.nl

E. Zambon
e-mail: zambon@cs.utwente.nl

– Those who want to get a general impression of the scope of graph transformation, and what a specification in this paradigm looks like;
– Those who want to get acquainted with GROOVE, and learn about its features and strengths;
– Those who already know graph transformation or GROOVE, and want to have some templates and examples of how to apply it in different contexts.

1.1 Background

Graph transformation has been advocated as a flexible formalism, suitable for modelling systems with dynamic configurations or states. This flexibility is achieved by the fact that the underlying data structure, that of graphs, is capable of capturing a broad variety of systems.

Essentially, whenever a system consists of entities with relations between them, this can be naturally captured by a graph in which the nodes stand for the entities and the edges for the relations. If, in addition, a main characteristic of such a system is that entities are created or deleted and the relations between them can change, then the transformation of those graphs comes into play.

A conceptual introduction to graph transformation can be found in [19]. The focus of this article is different. Rather than going into the theoretical background, we illustrate the uses of graph transformation on the basis of one particular tool that is capable of providing fast, hands-on experience, namely GROOVE (see [30]). We present four case studies from quite different domains, collected over the last 3 years, that show different features of graph transformation in general and of GROOVE in particular. Then, we more briefly review a number of other, previously published applications.

1.2 Introduction to GROOVE

In this section, we provide an overview of the features of the GROOVE tool. We describe the latest version, 4.0. Some of the older cases were developed with previous versions and consequently do not use all tool features, even where it would have been convenient to do so.

Graphs

Graphs in GROOVE consist of labelled nodes and edges.[1] An edge is a binary arrow between two nodes. Node labels can either be node types or flags; the latter can be used to model a boolean condition, which is true for a node if the flag is there and false if it is absent.

GROOVE can work either in an untyped or in a typed mode. In the untyped mode, graphs can be arbitrary: there are no constraints on the allowed combinations of node types, flags and edges. In the typed mode, all graphs and rules must be well-typed, meaning that they can be mapped into a special type graph. This is checked statically for the start graph and the rules: the theory then ensures that well-typedness is preserved under transformation. The type graph determines the allowed combinations of node types and edges.

Since typing is a new feature, only one case (Sect. 4) uses node types, and another (Sect. 5) uses full types.

Simple rules and application conditions

Graphs are transformed by applying rules. A rule consists of the following:

– A pattern that must be present in the host graph in order for the rule to be applicable;
– Subpatterns that must be absent in the host graph in order for the rule to be applicable;
– Elements (nodes and edges) to be deleted from the graph;
– Elements (nodes and edges) to be added to the graph;
– Pairs of nodes that are to be merged.

All these elements are combined into a single graph; colours and shapes are used to distinguish them. Alternatively, one may think in terms of *application conditions* and *modifications*: of the former, we distinguish positive (which must be present in order to apply a rule) and negative (which must be absent in order to apply a rule) ones, whereas of the latter, we distinguish deletion and creation of elements. Figure 1 shows a small example illustrating most of these concepts:

[1] Note that this is an extension: past versions of GROOVE only supported edge labels, node labels had to be mimicked by self-edges.

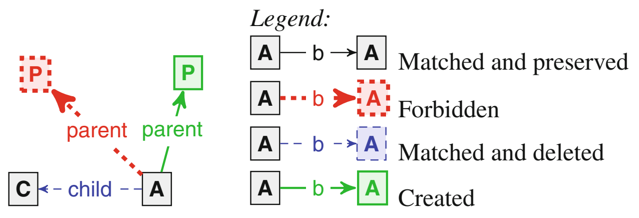


Fig. 1 Example GROOVE rule and legend

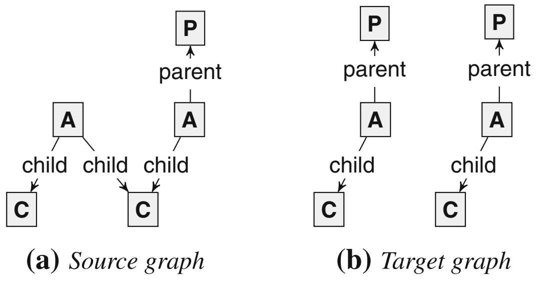


Fig. 2 Example application of the rule in Fig. 1

– The black (continuous thin) “reader” elements, in this case two nodes labelled **A** and **C**, must be present and are preserved—in fact, they form a positive application condition;
– The red (dashed fat) “embargo” elements, in this case a parent-labelled edge with a **P**-labelled target node, must be absent in the graph—in fact, each connected subgraph of embargo elements forms a negative application condition;
– The blue (dashed thin) “eraser” elements, in this case a child-labelled edge from the **A**-node to the **C**-node, must be present and are deleted;
– The green (continuous fat) “creator” elements, in this case a parent-labelled edge with a **P**-labelled target node, are created.

The overall effect of the rule is to search for **A**- and **C**-nodes connected by a child-edge but without a parent-edge to a **P**-node, and to modify this by removing the child-edge and adding a parent-edge to a fresh **P**-node. For instance, the rule can be applied to the graph on the left hand side of Fig. 2 in two ways, one of which results in the graph on the right hand side. (The other application removes the other child-edge.)

The core functionality of GROOVE is to recursively apply all rules from a predefined set (the graph transformation system) to a given start graph, and to all graphs generated by such applications. This results in a *state space* consisting of the generated graphs. The strategy according to which the state space is explored (e.g., depth-first, breadth-first or linear) can be set as a parameter.

Attributes

Nodes in a graph typically stand for instances of some resource or concept. For modelling most systems, however, it is also necessary to include *data* fields, containing booleans, integer numbers or strings. Such data fields are usually called *attributes*. GROOVE supports attributes by treating them as special edges that do not point to a standard node, but to a node that corresponds to a data value. Graphically such edges are usually represented by expressions of the form "x = 12", rather than by x-labelled arrows pointing to a 12-labelled node.

Regular expressions

Besides ordinary edges, a rule may include edges carrying regular expressions. These will be matched in the host graph by searching for a path whose labels satisfy the regular expression. This especially allows the specification of cycles or the transitive closure of edges.

Regular expressions may also contain *wildcards*, which are matched by any label in a given set. Moreover, wildcards may be *named*; such a name is effectively a variable for edge labels.

Quantification

One of the special features of GROOVE is the support of universal quantification in rules (see [33]). A universally quantified (sub)rule is one that will be applied to *all* subgraphs that satisfy the relevant application conditions, rather than just a single one as in the standard case. Such a rule can itself be much more concise, and also result in a smaller state space, than the equivalent set of rules that would ordinarily be needed. In fact, quantification can be *nested* in the sense that universally quantified rules can contain further existential subrules, and vice versa. Among other things, this makes it possible to formulate powerful application conditions (see [31]).

Control

The standard behaviour of GROOVE is to attempt the application of arbitrary rules at any point in time. There are, however, two further methods to control and direct the application of rules. A most straightforward mechanism is to assign *priorities* to rules: low-priority rules may only be applied if no higher-priority rule is applicable. A more sophisticated mechanism is to use GROOVE's *control language*.

A control program is imposed on top of a graph transformation system and specifies the allowed order of application of the rules of that system, referring to the rules by name. For instance, the control program `a;` **try** `{b;}` **else** `{c;}` specifies that first the rule named "a" must be applied, after which "b" is tried; if "b" is not applicable, "c" is applied. If rule "a" is not applicable in the beginning, then nothing happens. Other constructs offered by the language include:

- Looping, including an "as-long-as-possible" construct;
- A random choice between rules;
- Simple (non-recursive) function calls.

State space exploration

The most distinguishing feature of GROOVE, compared with other graph transformation tools (see Sect. 7 for an overview), is the fact that it does not just carry out a single sequence of transformations from a given start state, but can explore and store the entire state space of reachable graphs. This provides a rich source of information for further analysis. In fact, GROOVE offers a choice of the exploration strategy to be used:

- Depth-first full exploration, also with on-the-fly Linear Temporal Logic model checking;
- Breadth-first full exploration. In some grammars, this enables finding shortest paths to certain graphs;
- Linear, random linear, and conditional exploration. This allows simulation without covering all states, for instance if the state space is too large.

1.3 Structure

The remainder of this paper is structured as follows. In the next four sections, we describe four GROOVE case studies undertaken in the last few years:

- Section 2: Model transformation (from BPMN to BPEL);
- Section 3: Verification of a leader election protocol;
- Section 4: Analysis of security policies;
- Section 5: Simulation (modelling movements of ants).

For each of these case studies, apart from describing in some detail the actual solutions, we stress the special aspects of the problem and the GROOVE features used to solve it. In particular, Table 5 shows which GROOVE feature is used in which case study.

In Sect. 6, we briefly review a number of other applications of GROOVE in different domains. Finally, Sect. 7 contains an evaluation of the tool, based on experiences drawn from the case studies, along with a comparison between GROOVE and other tools.

2 Model transformation: from BPMN to BPEL

This case study presents an example of a model-to-model transformation. The source and target languages are BPMN and BPEL respectively, which conform to the model paradigm defined by OMG [2]. The task is to transform a standard representation of BPMN into a standard representation of BPEL.

Remark This case study was performed with a version of GROOVE that did not yet support typing. See also Table 5.

2.1 Case description

BPMN (defined by OMG, see [26]) and BPEL (defined by OASIS, see [27]) are languages for describing business processes. BPMN is a free-form graphical notation that is geared towards user-friendly modelling, and BPEL is a block-based notation that is geared towards transparent execution. This case study presents a solution in GROOVE for transforming (a subset of) BPMN to (a subset of) BPEL according to the transformation method described in [28,29].

This transformation problem was one of the challenges of the GraBaTs 2009 workshop [17] and the GROOVE solution [9] was one of the 10 solutions that were submitted. See the workshop homepage for a detailed description of all solutions.

An example of equivalent BPMN and BPEL models is shown in Fig. 3. It is taken from [10] and describes the process of publishing an article, which starts when the abstract is approved. The article is written and submitted to the editors. Then the writer waits for review results and submits a revised article, which is subsequently reviewed again. Depending on the result, the process ends or the article is edited and submitted to the editors again. The process ends with the publishing of the article. These models will be used as the running example in the remainder of this case study.

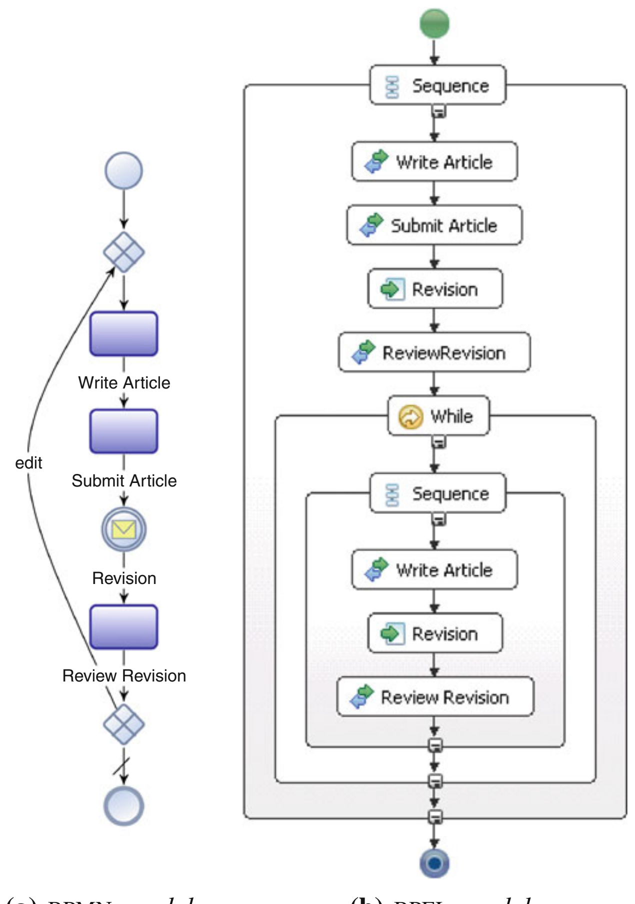


Fig. 3 Example model in BPMN and BPEL

2.2 Case features

The following features of this case study are of particular interest for this paper and for the application of graph transformation in general:

Model transformation

The translation of BPMN to BPEL is a model transformation that should make use of meta models as the description of allowed structure. This means that the GROOVE transformation should know about these meta models, and should operate on input that conforms to the BPMN meta model and produce output that conforms to the BPEL meta model.

Furthermore, the GROOVE translation should be aligned with the standard (file) representation for BPMN and BPEL models, which is by means of XML. By operating on XML, the input for the transformation can be exported from UML tools, and the output can be imported back again.

Deterministic transformation

The algorithm should behave in a deterministic manner: for each input BPMN model, it should always produce the same, uniquely determined BPEL model.

Control flow

The transformation algorithm used has a specific order in which the input BPMN model is traversed. It begins by recognising inner patterns, which are subsequently contracted into BPEL blocks. This then allows the recognition and contraction of bigger patterns. The algorithm works its way to the outer level until the model as whole has been transformed.

Due to the iterative nature of the algorithm, it is important that individual contraction steps can be tracked in a

[2] http://www.omg.org.

user-friendly manner. This allows the correct recognition of more complex patterns to be integrated easily into the algorithm, and ensures that problems in later stages can be debugged effectively.

Pattern matching

The power and usefulness of the transformation is measured by the number of different BPMN patterns it can transform to BPEL. Some patterns are more complex than others; for instance, a pattern with an arbitrary number of connecting paths is more difficult to recognise (directly) than a pattern with a fixed number of connecting paths. The number of patterns that can be recognised, as well as the ease of recognition, is a challenge for the graph transformation formalism.

2.3 Aspects of solution

The GROOVE solution for this case study is structured in the following five phases:

1. *Initialise*: translate BPMN (XML) to GROOVE (graph).
2. *Analyse gateways*: create explicit connections between opening and closing gateways that belong together;
3. *Analyse sequences*: mark the beginning and the end of sequences;
4. *Contract patterns*: transform recognised patterns (connected gateways) into BPEL blocks;
5. *Finalise*: translate GROOVE (graph) to BPEL (XML).

The 'wrapper' phases 1 and 5 are translations between XML and the internal (textual) representation of graphs that is used by GROOVE. These translations cannot be expressed within GROOVE, and have instead been realised by means of custom XSLT [42] transformations, which have to be carried out explicitly by the user. Two XSLT transformations for our particular case are specified in [9].

Phases 2, 3, and 4 are realised within GROOVE by means of graph transformation rules. In total, the translation consists of 46 rules (9 for phase 2, 16 for phase 3, and 21 for phase 4). The phases will be explained further below, using the BPMN model in GROOVE of Fig. 4 as a running example. Note that our representation is not structured according to a meta model or type graph, because GROOVE did not support those at the time.

Phase 2: analyse connecting gateways

In this phase, an explicit pattern edge is created between pairs of gateways that belong together. In other words, pattern recognition is performed, but without doing contraction. This is achieved with the following algorithm:

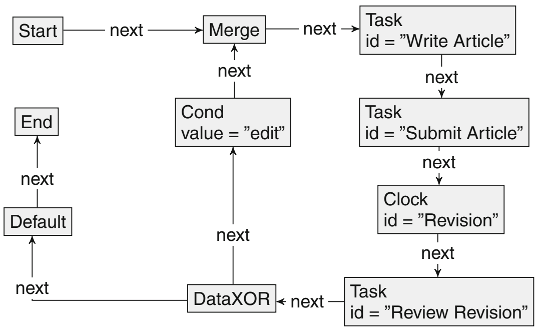


Fig. 4 Example model in GROOVE (after phase 1)

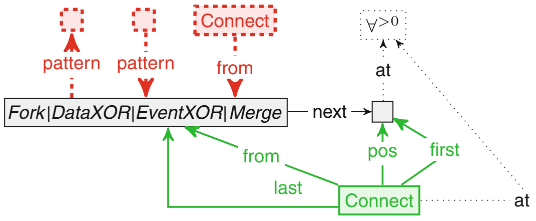


Fig. 5 The CreateConnects rule in GROOVE (phase 2)

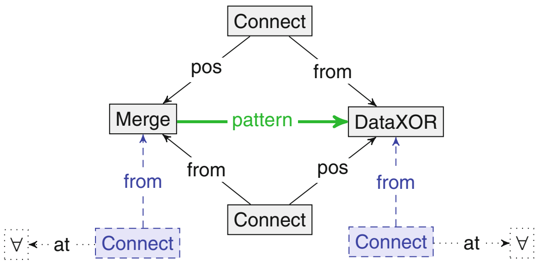


Fig. 6 The PatternWhile rule in GROOVE (phase 2)

1. For each outgoing edge next of each opening gateway, create a Connect node that establishes a link between the gateway and the target node of next;
2. Propagate Connect over basic units (and earlier recognised patterns) until a closing gateway is found;
3. Recognise patterns by analysing connections (i.e. if all Connect nodes from an EventXOR gateway lead to the same Merge gateway, then a Pick pattern was found).

Figures 5 and 6 show two of the rules of this algorithm. The rule in Fig. 5 creates the Connect nodes for a particular opening gateway. The non-vacuous universal quantifier node (labelled $\forall^{>0}$) ensures that a Connect is created for *each* next edge in one go, and also prevents the rule from matching if there is no next edge at all. The negative application conditions ensure that the rule will only be applied once for each opening gateway.

The rule in Fig. 6 detects a While pattern. It matches when the opening Merge and the closing DataXOR are connected by Connect nodes in both directions. After a successful match, the rule creates a pattern edge between the gateways,

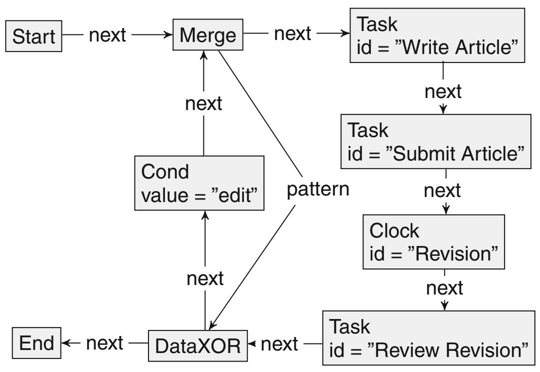


Fig. 7 Example model in GROOVE (after phase 2)

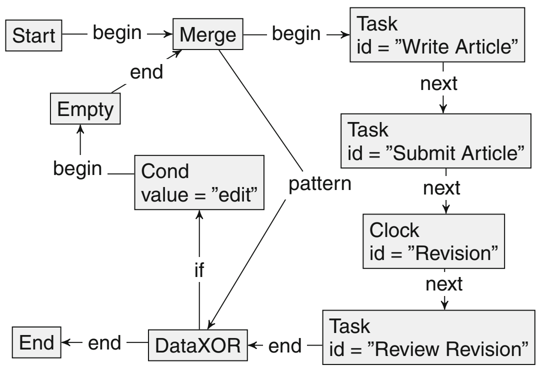


Fig. 8 Example model in GROOVE (after phase 3)

and destroys all Connect nodes that start from either the Merge or the DataXOR (again using a ∀ node).

The result of phase 2 on the running example is shown in Fig. 7. The main[3] point is the addition of the pattern edge.

Phase 3: analyse sequences

In this phase, all the next edges in the graph are renamed to better reflect their role. Edges leading to a 'block' (which is either the largest possible sequence or a single unit that is not part of a sequence) are renamed to begin, and edges leading out of a block are renamed to end. Also, some other cosmetic changes are made, including the renaming of edges in recognised Repeat pattern to if and the insertion of Empty node. In the example model, this leads to Fig. 8.

Phase 4: contract patterns

In this phase, the recognised BPMN patterns are contracted into BPEL blocks. Because the patterns have already been recognised in phase 2, and additional structural information is available by means of phase 3, contraction is mainly an administrative task only.

[3] In phase 2 also some additional administration, including the detection of so called quasi-patterns, takes place. In the example, this has resulted in the deletion of a Default node.

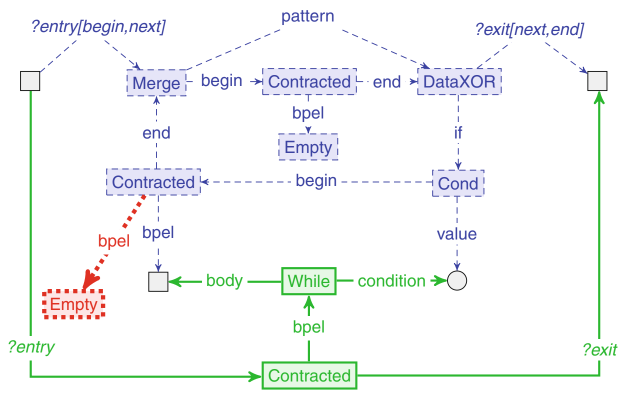


Fig. 9 The ContractWhile rule in GROOVE (phase 4)

For each pattern, the following actions are carried out:

1. A BPEL block node is created, and the incoming and outgoing edges of the pattern as a whole are redirected to it.
2. The paths between the opening and closing gateways are transferred to the newly created block node. This is only possible on paths that have already been converted, and, therefore, consist of a single BPEL block only.
3. All the remaining BPMN elements are removed from the graph, leaving only the BPEL block.

This is the same algorithm as in [28,29]. Note that the constraint that paths must already have been converted implies that contraction is carried out from the inside to the outer level.

In Fig. 9, a rule is shown which transfers While paths from a recognised DataXOR-Merge pair to BPEL nodes. The path is assumed to be Contracted, and its representation is changed to While. Note the wildcard label ?entry[begin,next], which matches on either begin or next (see also Sect. 1.2), and stores its match in the ?entry variable. This variable is then re-used in the rule to create an edge with the same label.

The result of phase 4 on the example model is shown in Fig. 10. This model can now be transformed into the final BPEL model of Fig. 3b with a XSLT transformation [9].

2.4 Evaluation

The transformation of BPMN to BPEL can be realised in GROOVE, with the help of wrapper XSLT transformations. The encountered strong and weak points of the use of GROOVE will be evaluated below in relation to the earlier introduced case features.

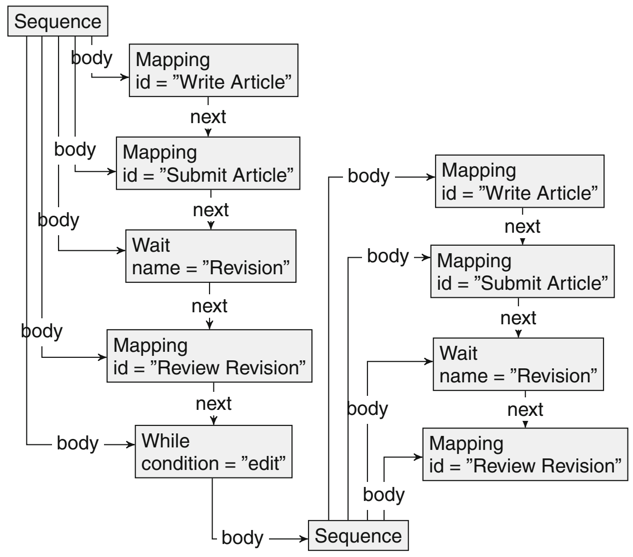


Fig. 10 Example model in GROOVE (after phase 4)

Using GROOVE *for model transformation*

The solution performs model-to-model transformation by translating a standard XML representation of BPMN to a standard XML representation of BPEL. This requires an external extension with XSLT, which is a *weak* point of GROOVE.

The solution does not make use of a meta model or type graph to define the structure of well-formed graphs. As a result, more effort was required, both for internalising BPMN and BPEL models, and for communicating the custom representation between different people working on the project.

This was a *weak* point of GROOVE when this case study was performed, because at that time there was no support for type graphs. However, this case study was one of the main arguments to add type graphs to GROOVE, and the current version of GROOVE no longer has this weak point.

Using GROOVE *for deterministic behaviour*

Graph transformation can give rise to non-determinism: it is allowed to apply rules in different orders, and the same rule can sometimes be applied in different ways. This behaviour is supported in GROOVE, as it builds a labelled transition system (LTS) in which *all* explored rule applications can be stored explicitly (see also Sect. 1.2).

In this case study, we are only interested in a single transformation path that leads to a single result. For this purpose, we constructed the system in such a way that the rules are confluent, which ensures that all paths converge. We then used the *linear exploration strategy* of GROOVE to compute a single path to the end result. Using this strategy, finding the output BPEL model can be performed by GROOVE without building a branching LTS, and takes less than one second for all the (small) examples in the case study.

Note that for (manually) checking that our rule system is confluent, we made use of the full state exploration of GROOVE. On all of the examples we tried, the full state space generated by our rule system converges to a single result.

Using GROOVE *for specific control flow*

The control flow of the algorithm was modelled in GROOVE mainly by rule priorities, which ensure that certain rules are always applied before others. In cases where this does not suffice, additional information was added to the graph which influences the enabledness of rules.

A specific *strong* point of GROOVE for building prioritised rule systems is its user-friendly interface, which allows the LTS to be inspected in many ways. The possibility to inspect the applicability of rules on each intermediate state of the state space greatly helps in determining the right rule priorities and the required rule interaction.

Using GROOVE *for complex pattern matching*

The realised transformation in GROOVE is able to identify arbitrary well-structured patterns (with an arbitrary number of paths between the opening and the closing gateway), as well as several quasi-structured patterns. This expressive power is mainly due to the separation of pattern recognition and contraction (which is an aspect of the solution), but the availability of quantified rules (see Sect. 1.2) in GROOVE is a contributing factor as well. Examples of quantified rules are shown in Figs. 5, 6. This case study uses quantified rules in a basic manner only; for a more elaborate use see Sect. 4. Still, the use of quantified rules is a *strong* point of GROOVE, also in this case study.

To summarise…

GROOVE can be used for expressing the BPMN to BPEL transformation, but as the tool is not specifically tailored for model transformation, several steps had to be carried out manually (such as the XML I/O). Also, the lack of type graphs (at the time of writing) lessened the ease of use.

Still, the transformation itself was not difficult to build, mainly due to the user interface (w.r.t. the LTS), the linear exploration strategy and the quantified rules.

3 Verification of a leader election protocol

In this section, we present a case study that illustrates how GROOVE can be used to verify communication protocols of distributed systems.

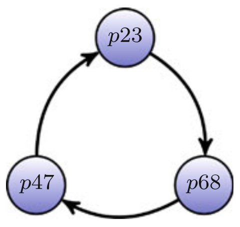


Fig. 11 A ring network of size three

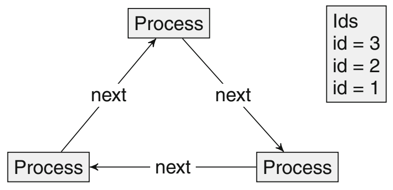
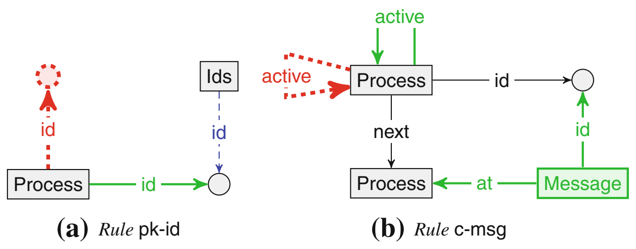


Fig. 12 The start state of a process ring of size three



(a) *Rule* pk-id **(b)** *Rule* c-msg

Fig. 13 Initialising rules of the leader election protocol

3.1 Case description

A simple distributed leader election protocol [5] works as follows. There is a set of processes arranged in a ring, i.e., every process has a unique predecessor and a unique successor. Furthermore, each process has a unique identity and there exists a total order over the set of identities (we assume that identities are natural numbers). The leader will be the process with the smallest identity; however, this information is not known at the start of the protocol.

Every process generates a message (*MId*) with its own identity and sends it to its successor. A received message with content *MId* is treated as follows by a process with identity *PId*:

– if $MId < PId$, the process forwards the message to its successor;
– if $MId = PId$, the process declares itself the leader;
– if $MId > PId$, the process discards the message.

A ring network with three processes is shown in Fig. 11. Each process has an identity and is connected to its successor.

3.2 Case features

The following features of this case study are particularly interesting.

Prototyping

The freedom that processes have in conducting different actions in different order due to the lack of a centralised controller leads to a high degree of parallelism. This level of parallelism is usually very hard to capture in models. Consequently, analysis and verification of the protocols are also very difficult. Therefore, a tool which enables the rapid prototyping of such systems can be very useful in the process of devising such protocols.

Verification

The main reason for modelling this case is to verify certain properties of the protocols for all different feasible scenarios which can occur as the result of different interleaving of events. To obtain this purpose, we need to generate the whole state space. Moreover, on the generated state space we need to verify both liveness, i.e., the protocol always declares a leader for all the configurations, and safety, i.e., never more than one leader is elected. Therefore, all generated paths need to be checked for these properties.

General rules

The same protocol in this case should work for rings with different sizes. In other words, the size of the start graph can be chosen parametrically while the set of rules stays unchanged.

3.3 Aspects of solution

Figure 12 shows the start graph modelled in GROOVE. It consists of an example network with three processes modelled as nodes connected in a ring topology. There is an extra auxiliary node Ids containing identities ranging 1–3. This node is used to generate all possible permutations of processes with different identities in the network. The graph can easily be extended for any arbitrary number of processes. Note that the selection of identities among numbers from 1 to n can be regarded as a canonical representation of any arbitrary sequence of n identities.

As seen in Fig. 12, initially processes do not have any identity assigned to them. Rule pk-id, shown in Fig. 13a, has the highest priority, and it assigns identities to the processes before any other rule can be applied. GROOVE automatically makes all possible non-deterministic choices of all applicable rules in generating the state space. In this way, we generate all different permutations of identity assignments as required for the protocol verification. Rule c-msg, shown in Fig. 13b, creates the initial messages and marks the processes as active

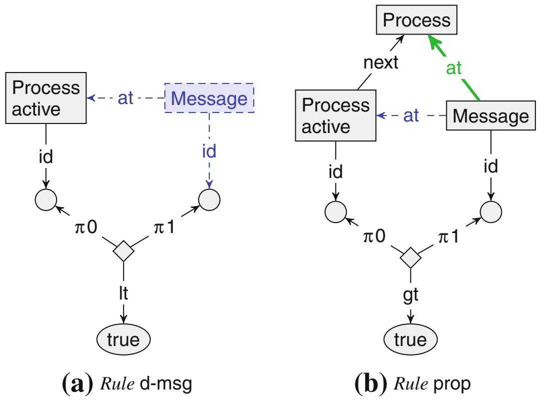
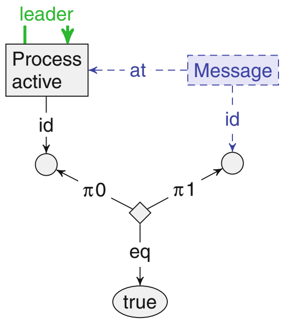


Fig. 14 Message relaying rules of the leader election protocol

Fig. 15 Electing leader rule of the protocol



to avoid the sending of more than one initial message per process.

Fig. 14 shows rules d-msg and prop. The d-msg rule discards messages whose identity is higher than the identity of the receiving process. Finally, rule prop relays a message on a process if the identity of the message is smaller than that of the process. Finally, rule elect shown in Fig. 15 elects the leader when a process receives a message with its own identity.

The state space of the protocol is obtained by applying the rules shown in Figs. 13, 14 and 15 on the start state of Fig. 12. The state space is shown in Fig. 16 as a *labelled transition system* (LTS) in GROOVE. States are displayed as rectangles and the names of the rules applied for transformations from one state to the other are written as labels of the transitions between states. The top state ($s1$, in green) shows the starting state and the bottom states ($s66$ and $s70$, in red) show the final states, namely, no other state can be reached from these states by applying any rule.

The state space consists of two parts. In the upper section (the states above states $s11$ and $s12$) only rule pk-id is applied to create the ring. The lower section is where the rules related to the protocol itself are applied. A worthwhile point about the upper part of the state space is that it creates all permutations of a network with size n. For a network with size n, there are $n!$ different permutations of nodes which determine the different number of orderings in which rule pk-id can be applied on the nodes. However, we know that due to the

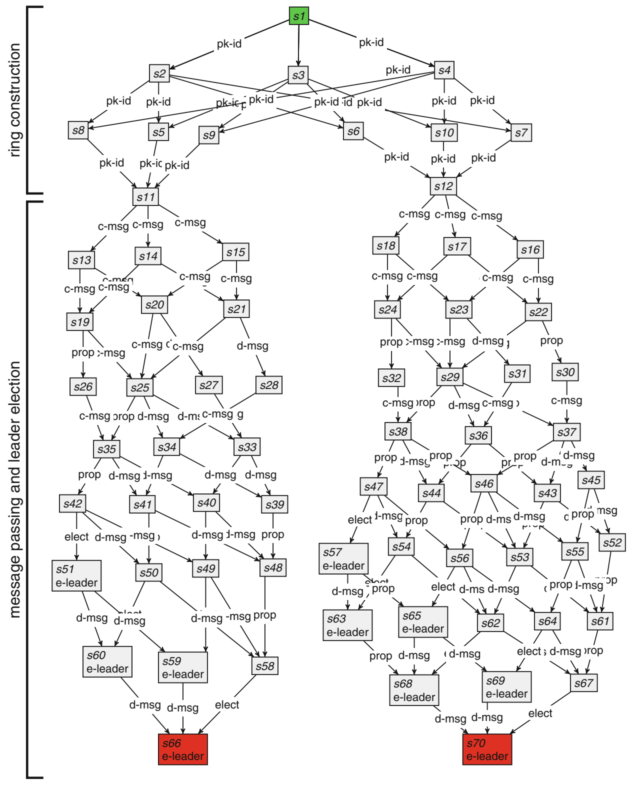


Fig. 16 The LTS of the protocol

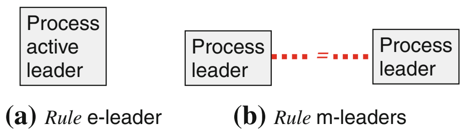


Fig. 17 The verification rules

symmetry of a ring, only $(n-1)!$ different rings exist with the same set of nodes. Not detecting the identical states in this case leads to a state space that is almost n times bigger. GROOVE automatically finds the identical states (isomorphic graphs) and avoids duplicating already existing states. This can be seen in Fig. 16, all six (3!) different feasible orders of applications of pk-id are shown in states $s5$ to $s10$, which are reduced to two states: $s11$ and $s12$. The protocol is only checked on these two generated rings.

3.4 CTL model checking

GROOVE allows us to verify properties specified in CTL (Computation Tree Logic). To verify the generated LTS, we add two rules to assist us with the model checking part. Figure 17 shows these rules. The liveness property holds if the rule e-leader is applicable. The safety property is only true if the rule m-leaders is not applicable.

The liveness property is preserved if we have no counter-example to AF(e-leader), meaning that all paths in the the

Table 1 Experimental results for leader election protocol

Processes	Rings	States	Transitions	Time (s)
3	2	70	147	<1
4	6	677	1,790	<1
5	24	9,358	30,457	6
6	120	168,422	656,214	66
7	720	4,747,432	23,914,934	4,058
8	5,040		Out of memory	

LTS eventually lead to the choice of a leader. The safety property can be verified if there is no counterexample to $AG(!\text{m-leaders})$, which means that there should not be two different leaders in any state of the LTS.

3.5 Experimental results

We tested our rule set for rings with three to eight processes. For our tests we used a machine with two Quad Core Xeon 2.66 GHz processors and 16 GB memory.

The results are shown in Table 1. The first two columns show the number of processes (size of the ring) and the number of different configurations for the rings of the given size, respectively. The results of the experiments are given in three columns. The first two columns denote the number of states and transitions in the generated transition system and the third column shows the total amount of time used for both the state space generation and the verification. We have verified our results using the formulae explained in Sect. 3.4.

3.6 Evaluation

In this section, the strong and weak points of the given solution are discussed in relation with the case features explained in Sect. 3.2.

Using GROOVE *for prototyping*

We have modelled a leader election protocol which contains a high degree of parallelism. The simplicity of the rules in the proposed solution shows that the modelling phase was intuitive. Part of this simplicity is due to the absence of typing. In this case, the use of a type graph is not necessary and the solution does not benefit from it. Besides, all rules in GROOVE are visual which is very useful in this case study. In [17], more elaborate solutions were proposed, on three different variants of the protocol, which is a good evidence on how easy different variations of a problem can be modelled and analysed.

Using GROOVE *for verification*

For the given experiments, the whole state space was generated and both liveness and safety properties were verified using CTL formulae. No counter example was found for our experiments which proves the correctness of the protocol for rings with size smaller than eight nodes. Furthermore, no assumptions were imposed on the message relaying (except that each process has a buffer of size n). Hence, we have verified that the protocol works regardless of the buffer policy adopted (e.g., FIFO, LIFO, etc). This is a very interesting general result.

General solution in GROOVE

As seen from the solution, the same rule system works on networks of different sizes. But the whole verification process is not general for a ring of an arbitrary size. However, only the start graph needs to be adapted for any given ring size, which can be done with little effort, while the rest of the rule system stays intact.

To summarise…

GROOVE can be used as a rapid prototyping tool which is easy to use in all three phases of modelling, analysis and verification. As the result, GROOVE can provide a great assistance in devising network protocols, where non-determinism as well as parallelism is an essential parts. However, the problem does not scale in GROOVE for problems with large sizes. This is because the size of the state space grows dramatically as the ring size increases. This is the well-known state space explosion problem, common to all model checking tools.

4 Analysis of security policies

In this section, we present an organisational security framework and describe how GROOVE can be used to model and analyse security properties within such a framework.

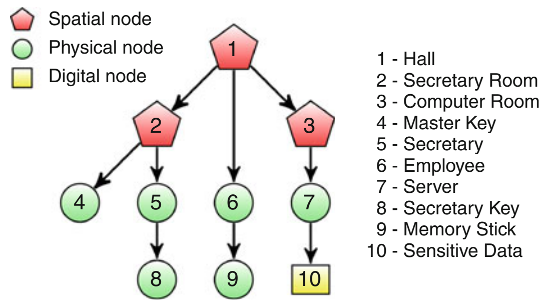


Fig. 18 Example of an environment graph of the Portunes framework

Table 2 Containment relation for the environment graph of Fig. 18

$\succ_{ln}$	1	2	3	4	5	6	7	8	9	10
1 - Hall				1	1	1	1	1	1	
2 - SRoom				1	1	1	1	1	1	
3 - CRoom				1	1	1	1	1	1	
4 - MKey										
5 - Sec				1				1	1	
6 - Emp				1				1	1	
7 - Server									1	1
8 - SKey										
9 - MStick										1
10 - SData										

4.1 Case description

The Portunes security framework (developed by Dimkov, Pieters, and Hartel [12]) has two main goals: (1) to define a unified model that captures the relations between physical, digital and social security domains, and is able to describe security attacks which span these three domains; and (2) to provide analysis techniques to detect security breaches in the environment of an organisation.

Environment graphs

Portunes uses an *environment graph* as a snapshot of a certain configuration of the organisational environment. The graph stratifies nodes in three layers. The spatial layer is formed by the facilities of the organisation, e.g., rooms and halls. The physical layer consists of objects located inside the facilities, such as people, computers, and keys. The digital layer comprises software and data, such as operating systems and databases.

Figure 18 shows an example of an environment graph in Portunes where membership in each of the three layers is identified by different node shapes. Edges between spatial nodes represent neighbourhood. In this example, we have a hall (node 1) that connects two other rooms (nodes 2 and 3). All other edges represent a containment relation. For instance, we have a secretary room (2) that contains the master key (4) and the secretary (5), which in turn is in possession of her own key (8). For brevity, we will refer to nodes in the environment graph with simple abbreviations of the names given in Fig. 18.

Containment relation

The environment graph presented in Fig. 18 is one element of a set of possible entity configurations. Some of these entities (graph nodes) may have active behaviour. For example, people can move around rooms and can exchange objects that they are carrying. To capture this dynamic behaviour it is necessary to transform one given environment graph into another, by removing or adding containment edges. In order for these containment changes to make sense in reality, they must obey a containment relation $\succ_{ln}$ that defines whether a node can contain another node or not.

The containment relation for our example is given in Table 2, as a boolean table. A value of 1 indicates that the row element can contain the column element. A value of 0, presented in Table 2 as an empty cell, shows that such containment is not possible. Relation $\succ_{ln}$ is not symmetric, e.g., we have that MStick can contain SData but not vice-versa.

It is important to note that relation $\succ_{ln}$ is used only to represent containments that are feasible in reality. This relation, however, does not enforce security policies.

Actions and access control policies

An *action* is a primitive that manipulates nodes of an environment graph. Portunes defines three basic actions:

- *Login*: which allows a node to "enter" another, i.e., the action adds a containment edge to the graph;
- *Logout*: which allows a node to "leave" another, i.e., the action removes a containment edge from the graph;
- *Eval*: which allows a node to "delegate" an action to another node.

In the framework definition [12], the operational semantics of these basic actions is formalised by a set of inference rules.

Each node of the environment graph has one or more access control policies that allow the execution of a subset of basic actions, provided that the acting node has the proper security privileges. An access control policy is composed by three ways of authentication:

- *Identity based*: meaning that the acting node must have the required identity (i.e., name) for the action to be allowed;
- *Location based*: meaning that the acting node must be in the required location for the action to be allowed;

– *Credentials based*: meaning that the acting node must possess all elements of a set of credentials (physical or digital nodes) for the action to be allowed.

Here, we informally describe the security policies for some of the nodes of the environment graph of Fig. 18. In the next section, we show how these policies are modelled in GROOVE (see Fig. 21).

– SRoom. To enter the secretary room: (a) no identity is needed; (b) the actor must be in the hall; and (c) the actor must have either the secretary key or the master key. To leave the secretary room it suffices to be inside it.
– CRoom. The security policies for node CRoom are similar to SRoom with the exception that it is only possible to enter the computer room with the master key.
– Server. The policy for node Server is location based, and it states that it is only possible to login or logout in the server from the computer room.
– Sec. The policy for node Sec is identity based and it defines that the secretary allows the employee to perform any basic action.

Attack scenarios

Let us assume in our running example that the employee has malicious intentions and wants to copy the sensitive data stored in the server to his memory stick. Initially, he is not able to do it, since he does not have the proper credentials (the master key to open the computer room). However, there is at least one sequence of actions that allows him to accomplish his goal. This constitutes an *attack scenario*.

A textual description of a possible attack scenario is as follows. The secretary moves to the hall. There, the employee asks the secretary to borrow her key, for example, to pick office supplies in her room. Since the secretary trusts the employee (her access policy defines this), she lends him her key. The employee then goes to the secretary room, retrieves the master key, moves back to the hall and returns the secretary key to her. After she returns to her room, the employee is able to enter the computer room and copy the data from the server to his memory stick.

In the attack scenario just described, no security policy is violated when the attack is performed. In our simple example such security breach can easily be discovered, but in more complex environments this task is far from trivial.

4.2 Case features

For the framework to be useful in practise, its implementation should provide the following desired features.

Automatic generation of scenarios

The interleaving of actions from different nodes gives rise to a huge amount of non-determinism, which renders a manual search for security breaches unfeasible. Tool automation is, therefore, necessary to systematically search (i.e., generate) attack scenarios. An additional important feature is the possibility to simulate an attack in a step-wise fashion, to allow the user of the framework to reproduce and analyse the generated scenarios.

Scalability

The environment graph describing an organisation can be formed by hundreds or even thousands of nodes. The framework implementation should properly scale to such graph sizes, providing results within a reasonable time limit.

General solution

Security policies differ from one organisational environment to another; however, the mechanism to enforce such policies is described in the framework by general inference rules. Implementation of such rules should preserve their general property, to avoid the need to define specific security enforcement rules for each environment analysed.

Diverse audience

The framework is intended to be used by a broad audience, with different backgrounds, e.g., security consultants and company managers. To ease the understanding of the functionality of the implementation, only elements of the security domain should be visible. In particular, the theoretical intricacies of the framework do not concern the users, as long as the resulting analysis is sound.

4.3 Aspects of Solution

The GROOVE solution for this case study, i.e., our framework implementation, is elaborated in the following manner:

1. A Portunes environment graph, the containment relation $\succ_{ln}$, and the environment security policies are all modelled in a GROOVE state graph.
2. The behaviour described by Portunes operational semantics, the enforcement of security policies, and the possible actions of active nodes are defined by transformation rules in GROOVE.
3. Attack scenarios are generated by performing state space exploration in GROOVE.

In the following, we discuss each of these items in detail.

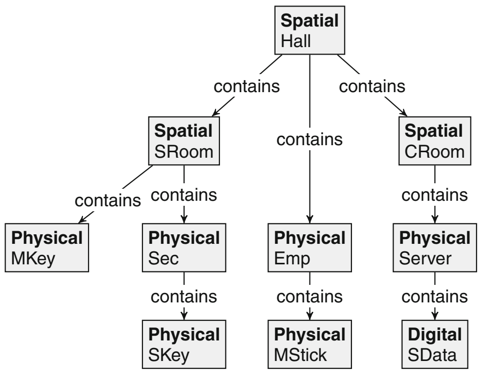


Fig. 19 The environment graph of Fig. 18 represented in GROOVE

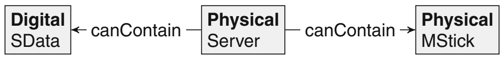


Fig. 20 Sample of the containment relation $\succ_{ln}$ represented in GROOVE

Environment graphs in GROOVE

The mapping of Portunes environment graphs to GROOVE is trivial. Fig. 19 shows the GROOVE counterpart of the environment graph of Fig. 18. Nodes are identified by a proper unique label that represent the entity, e.g., MKey. In addition, each node has a proper type label, shown in bold in Fig. 19, that encode the meaning of the geometric shapes used in Portunes environment graphs.

In Portunes, edges have different meanings depending on the nodes they connect, but this meaning is implicit in the framework. In GROOVE all edges require a label. Instead of using different labels to represent neighbourhood and containment, all Portunes edges are encoded in GROOVE as edges labelled contains. This uniform representation does not invalidate the modelling and simplifies the design of transformation rules.

Containment relation in GROOVE

The relation $\succ_{ln}$ is encoded in GROOVE by edges labelled canContain. For every two nodes for which $\succ_{ln}$ holds, an edge is added to the GROOVE state graph. Figure 20 shows the encoding of line 7 of Table 2 in GROOVE.

Access control policies in GROOVE

The security policies of each entity are encoded in GROOVE with additional policy nodes and edges. Each spatial, physical or digital node has one or more outgoing policy edges that define its policies. A policy edge goes to a policy node that lists the actions allowed by the policy (*login* – In, *logout* – lt, or *eval* – e) and specifies the security requisites of these actions.

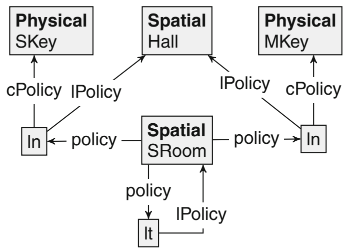


Fig. 21 Secretary room security policies represented in GROOVE

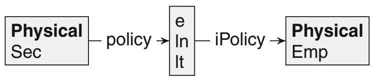


Fig. 22 Secretary security policies represented in GROOVE

Figures 21 and 22 depict the security policies for the secretary room (node SRoom) and the secretary (node Sec), presented as text in the previous section. The room has three access polices, two for login and one for logout, and the secretary has a single policy for all actions. The requisites for identity, location and credential based access are identified by edges labelled iPolicy, lPolicy, and cPolicy, respectively.

Actions in GROOVE

Describing the behaviour of active nodes using only the basic actions defined in Portunes is cumbersome. However, more elaborate actions can be constructed from the basic ones. In our solution, we use the following high level actions:

- *Move*: a node can move either up or down in the containment hierarchy of the environment graph;
- *Pick*: an active node can pick an inactive one that is in the same location;
- *Request*: an active node can ask another node in the same location to give up one of its possessions;
- *Spawn*: an active node can temporarily activate one of its possessions and place it under an inactive node;
- *Merge*: an active node can deactivate and reacquire its temporarily active nodes.

A node can perform a *move* action when the containment relation is satisfied and the security policies allow the node to *logout* from its source and *login* in its destination. In all other actions, the same conditions for *move* also apply. Additionally, in the *request* action, the security policies must also allow a node to perform *eval* at its target.

Each of the high level actions just described is implemented in GROOVE by a sequence of rules. To simplify the

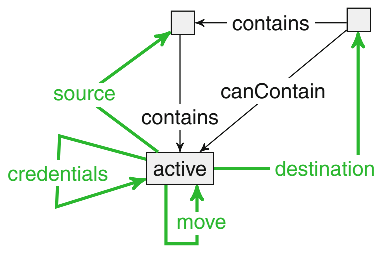


Fig. 23 Rule that describes the intention of an active node to move up in the environment

design, we divided the rules into two groups: the ones who describe behaviour and the ones who enforce security. The behavioural rules describe the intention of an active node to perform a certain action. Such behavioural rules are always followed by one or more security rules that check the security requisites of the acting node and perform the action when allowed.

Figure 23 shows a behavioural rule where an active node wants to move up in the environment. This is indicated by the new move edge, along with the edges that mark source and destination of the node and its security credentials. The security polices are checked by the rule depicted in Fig. 24, which also performs the move, when the policies allow. The central node is the one moving, as pointed by the move edge. In order to change its location, the moving node must satisfy the logout policy of its source node and the login policy of its destination node. These policies are identified in Fig. 24 by the nodes labelled lt and ln, respectively. We focus on the lt policy at the left; the ln policy is very similar. In order to capture the fact that a policy may require several credentials, we use nested quantifiers, where the existential level is named, to allow proper identification of required edges. For example, the leftmost part of the rule in Fig. 24 states that all credentials specified in the logout policy of the source node should be contained in the credentials of the moving node. In this case, the existential level is named c1 and corresponding containment edge is pre-fixed with this name. Identity and location policies are handled in the same way, with their existential levels named i1 and l1, respectively.

To ensure that a behavioural rule is always followed by a security rule, we use the control functionality of GROOVE. A high level action (e.g., *move*) is defined by a function in the control program, and this function is composed by a sequence of rule applications. A more detailed example of the use of a control program in GROOVE is given in Sect. 5.

Attack scenarios in GROOVE

The generation of an attack scenario is done in GROOVE by means of a state space exploration. In order to guide the exploration, we define an additional rule that describes a specific security breach that should not occur. Figure 25 presents this rule for our running example. We have a security breach when the employee manages to reach the hall in possession of the memory stick with the sensitive data.

GROOVE has an exploration option which performs a breadth-first search until an application of a certain rule is found. Using this option with the breach rule we instruct GROOVE to search for the shortest attack scenario, i.e., a trace from the breach rule application to the start state. This trace can be highlighted in the LTS and each of its steps inspected in GROOVE. If the exploration terminates without a trace being found, we can assert that the security policies prevent the given breach. For our running example, the shortest attack trace found by GROOVE is formed by 22 rule applications.

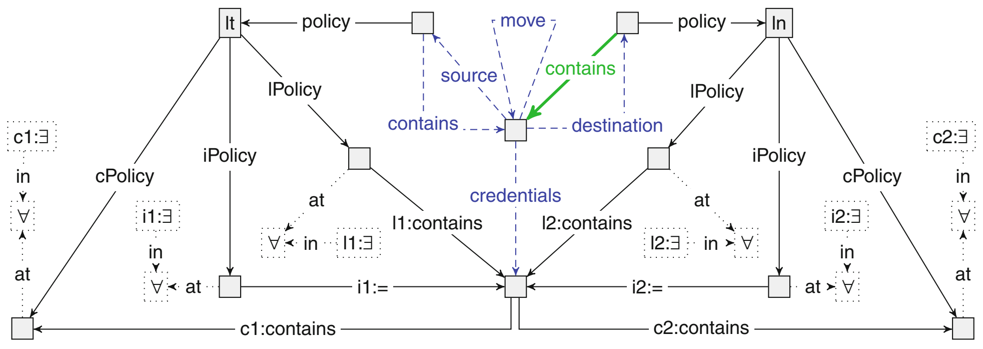
4.4 Evaluation

The functionality of the Portunes framework were properly implemented using GROOVE. In the following, we give a more specific evaluation of each of the case features.

Using GROOVE *to automatically generate scenarios*

The ability to perform state space exploration of graph production systems is one of GROOVE's strongest points and it

Fig. 24 Rule that checks security credentials and perform a move action



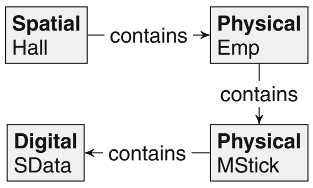


Fig. 25 Rule that defines a security breach

makes the tool very suitable for this case. The possibility to guide the exploration to search for a specific security breach is of particular interest.

Scalability of GROOVE *solution*

The implementation of Portunes that we present in this section is a proof-of-concept, designed to be simple and easy to understand. In particular, a point that was not properly addressed is scalability. Tests show that performance degrades fast when start graphs grow to more than hundreds of nodes. To tackle this problem, Dimkov et al. [11] developed a more elaborated GROOVE solution, with a similar modelling of environment graphs but a different set of transformation rules. To further improve the performance they are now developing exploration algorithms tuned to this particular case and will extend GROOVE to their own needs. This is possible because GROOVE is open source software and provides an externalisation API that allows the tool to be extended in a simple way.

Generality of GROOVE *solution*

All rules defined in our solution are general, in the sense that they are not tuned to a certain environment graph. This means that the same set of rules can be used in every analysis, and the user only needs to provide the start graph that describes the initial configuration of the environment and the rule that constitutes a security breach.

A feature of GROOVE that permits such a general solution is the use of nested rules. A nested rule allows changes to be made to sets of sub-graphs at the same time, rather than just to the image of an existentially matched LHS. An example of a nested rule is shown in Fig. 24.

The use of nested rules allows complex actions to be expressed neatly within small rules. Without quantifiers, the rule would have to be split in a constant part and a to-be-repeated part. Also, it may be necessary to explicitly add control to the part that must be repeated, to ensure that its beginning and its end can be detected statically. Since all transformations specified in nested rules are performed in one transition, this type of rule also reduces the state space of the graph transition system. Having nested rules is a clear *strong* point of GROOVE.

Diverse audience of GROOVE *solution*

Tools based on graph transformation usually have a strong visual appeal. The graphical interface of GROOVE offers a large set of visual capabilities that are both powerful to use and easy to master. In this case, we believe that the graphical visualisation of rules is an improvement over the original inference rules of Portunes. Furthermore, the tool keeps all the theoretical aspects of graph transformation under the hood, and presents a simple "push-button" interface. This helps to capture the interest of a larger group of users, as witnessed in a masters' course on security, where Portunes and GROOVE were used in teaching.

To summarise…

The dynamic behaviour of entities can be easily modelled in terms of graph transformations, making GROOVE an adequate tool for this case study. The GROOVE functionality highlighted in this case are: guided state space exploration, nested transformation rules, and graphical user interface. The issue with scalability of the solution is being tackled by the creators of Portunes.

A detailed description of the framework can be found in [12], and the implementation is available from the Portunes project in SourceForge [11].

5 The AntWorld case study

This case exemplifies the use of GROOVE as a tool for prototyping the behaviour of a given system. The system itself is a synthetic benchmark, used in the GraBaTs 2008 transformation tool contest (see [34]).

5.1 Case description

The AntWorld simulation consists of an ant hill sitting in the middle of a large area. The ants are moving around searching for food. If an ant finds food, it brings the food home to its ant hill to grow new ants. On its way home, the ant drops pheromones marking the path to the food reservoir. If an ant without food leaves the hill or if a searching ant hits a pheromone mark, the ant follows the pheromone path to the food. This behaviour should result in the well-known ant trails.

The area grid

The area in which the ants move consists of a grid of nodes. In order to enable the ants to go home on a straight path if they have found some food, the area grid shall look like a spider's web, cf. Fig. 26. (In this figure, the ants have not yet found

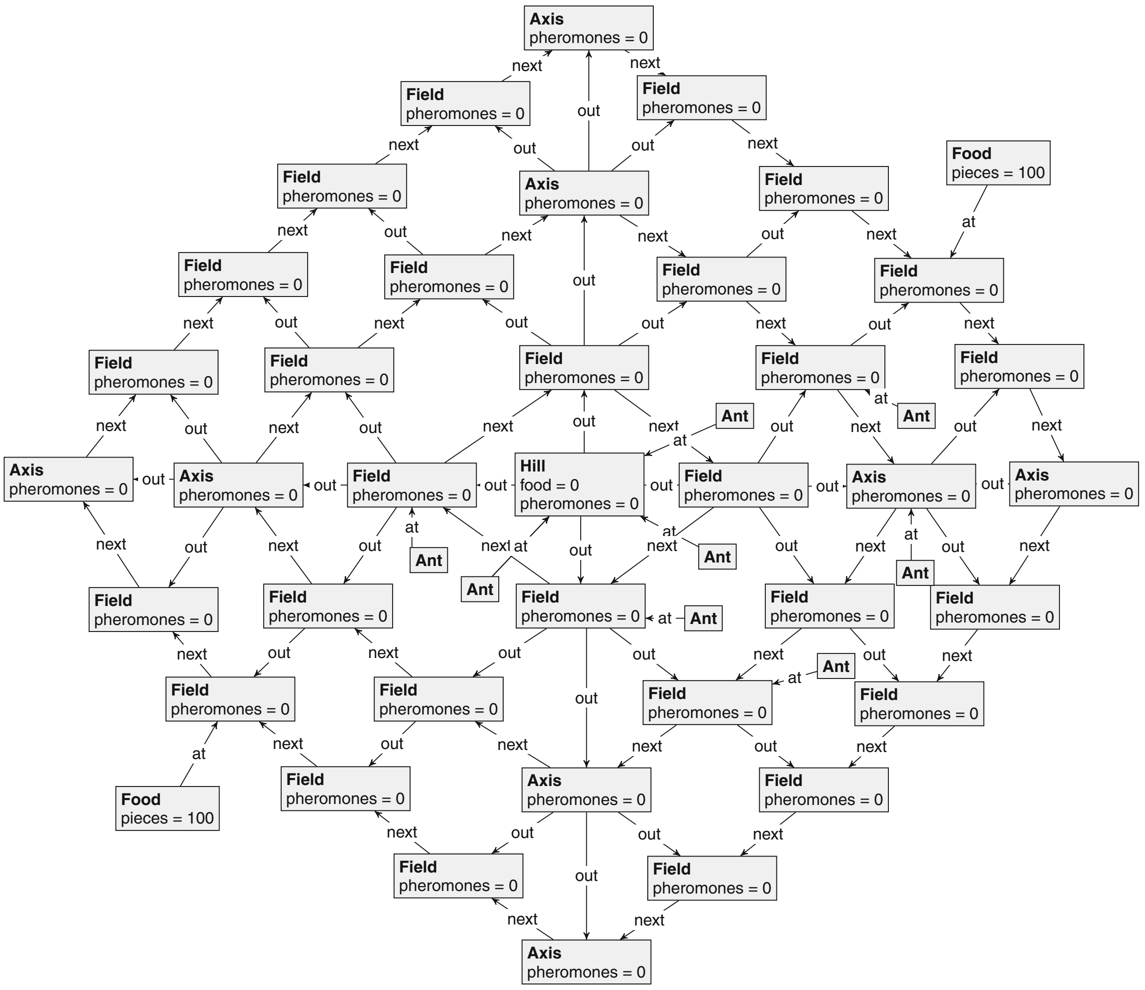


Fig. 26 Example layout for the ant hill's area grid

any food; consequently none of the fields have associated pheromones.)

The AntWorld simulation works in rounds. Within each round, each ant makes one move. Afterwards, the area may expand, pheromones may evaporate, and new ants may be born.

- Initially, the area grid consists only of the hill and the first two circles.
- Whenever an ant enters the currently outermost circle (i.e. the border of the yet known area), a new circle of nodes is "discovered" and should be created. Every 10th node of this new circle carries 100 parts of food.
- After each round, 5% of the pheromones on each field evaporate; moreover, the hill creates one new ant per delivered food part.

Ant moves

The ant behaviour depends on the following modes:

- An ant without food is in search mode. It either takes one piece of food (if it finds one) and enters the food carrying mode, or it goes to a randomly chosen neighbour field, favouring those with enough pheromones;
- An ant in food-carrying mode follows the links towards the inner circle, dropping pheromones as it goes along (which guides other ants to the food place), or drops the food on the ant hill and enters search mode again.

Goals

The goals of the case study were as follows:

- Tools shall model and run the AntWorld according to the above rules.
- For performance measurement, tools shall report, for reasonable numbers of rounds, the number of circles of the grid, the number of ants created, and the total execution time.
- If possible, tools shall provide animations showing the ants and how they search for food and form ant trails.

5.2 Case features

The following features of this case study are of particular interest.

Prototyping

The central problem of this case study is to encode the behaviour of a particular system—in this case, the system of an anthill, as formulated in a number of rules. Thus, the case study tests the ability of a modelling environment to provide a model that faithfully encodes the given rules.

Control flow

Part of the problem of prototyping is that the rules of ant movement and the extension of the area grid are complex; in fact, they are composed of multiple steps, or phases. The easiest way to model them faithfully is to make this composition explicit, using some form of control flow.

Simulation

The purpose of the prototype is to study the emerging behaviour of the ant colony. For instance, the model should show the formation of ant trails seen in nature. For this purpose, the most important feature required of the modelling environment is the ability to actually simulate the rules, either in a single-step mode or in an automatic multi-step mode, and observe their effect. Moreover, the behaviour of the ants is highly non-deterministic; the simulation should be able to reflect this, in the sense that different runs should result in different outcomes.

Animation

The simulation goal clearly states the desirability of providing animated behaviour of the system.

Performance

For the (multi-step) simulation to give rise to interesting results, the more steps it consists of the better; also, for a reasonable coverage of possible behaviours it is important to rerun the simulation. Both factors require a high performance of the simulation environment. Note that the case has the interesting characteristic that the system (the ant hill) never stops growing; this will have a negative impact on performance.

5.3 Aspects of solution

Since the case description is already in a rule-based format, modelling is relatively easy: essentially, we need to develop a graph representation on which all rules should be made to work. However, some aspects require a bit of care.

The process that we have followed in arriving at the solution consists of several steps. In the first step, we prototype the desired behaviour as directly as possible, without considering the performance of the resulting rules and without using type checking. This is refined in further steps: large performance gains can be obtained by reducing non-determinism where possible, by avoiding regular expressions, and by guiding the search plan. Typing is useful for documentation and maintenance, but (in the current implementation) does not speed up the simulation.

Rounds and phases

To ensure that the system displays the required behaviour in rounds, and within each round goes through the prescribed steps, we need to restrict the applicability of rules: at any given stage of the simulation, only a limited number of rules (viz., only those to do with the active phase) should be enabled. In order to achieve this, we use the following control program:

This will execute the function `main` until stop matches. `main` directly reflects the phases in the case description: ants reproduce, ants move (in the function `turn`), pheromones evaporate, the area grid may grow (in the function `grow`), and finally the turn counter is increased.

The function `turn` specifies that, as long as a new ant can be selected (as determined by the turn-edge, which is updated to the next round as soon as an ant is selected), this ant can attempt to drop or pick up food, after which it makes a move through the grid. In fact the rule select_ant attempts to find an ant that has not yet moved in this turn, and marks it as selected; the other rules in `turn` match the selected ant, and the move_*-rules deselect it. Some of the rules are shown in Fig. 27.

Note that move_random uses a regular expression to specify that movement may occur backwards or forwards along out- or next-edges.

For another rule, evaporate, the control program does not show a loop, even though the rule should be applied to all fields with a positive number of pheromones. This is because this behaviour lends itself to be specified using a quantified rule, shown in Fig. 28.

Typing

Figure 29 shows the type graph for the GROOVE AntWorld solution. Without going into all the details, we mention one

```
until (stop) { main(); }

function main() {
        alap { reproduce; }
        turn();
        evaporate;
        if (on_edge) { grow(); }
        end_turn;
}

function turn() {
        while (select_ant) {
                try { drop_food; }
                else try { pickup_food; }
                try { move_home; }
                else try { move_search; }
                else { move_random; }
        }
}

function grow() {
        expand_start;
        until (expand_end) {
                try { expand_axis; }
                else { expand_normal; }
        }
        put_all_food;
        cleanup_index;
}
```

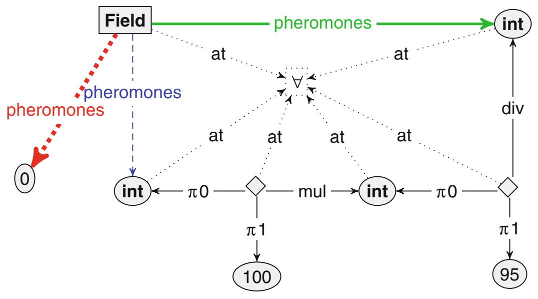


Fig. 28 5% of all pheromones evaporate

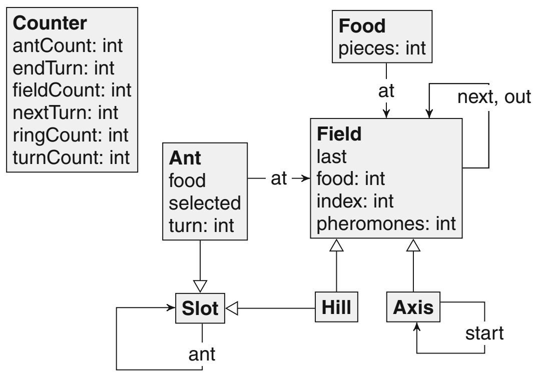


Fig. 29 Type graph for the AntWorld solution

interesting aspect, namely that GROOVE supports multiple inheritance: **Hill** is a subtype both of **Field** and of **Slot**. The latter is used to control the ordering of the ants, which we use to select them deterministically; see below.

Non-determinism

Matching and applying transformation rules are inherently non-deterministic. Non-determinism in principle affects the performance adversely, because the search for rule matches takes more time. In this case, the non-determinism partly reflects the problem that is modelled, but other parts of the system behaviour are completely deterministic. For instance, the function `grow` in the control program, which has the effect of extending the grid with an additional ring, is triggered non-deterministically (by the movement of an ant) but, once triggered, can proceed completely deterministically. In fact, the corresponding rules (not shown here) have been constructed in such a way that their effect is deterministic.

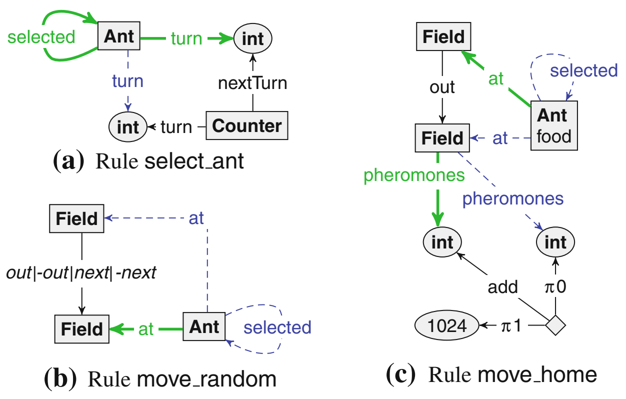


Fig. 27 Selection and move rules

The ant movement, on the other hand, is supposed to be non-deterministic, and in fact should vary across simulation runs. This is achieved by the so-called *random linear* evaluation strategy of GROOVE: at every state, a single random choice is made between enabled rule applications, after which simulation proceeds at the new target state. This is, therefore, quite different from the full state space exploration in the leader election case (Sect. 3).

However, let us also consider the order in which ants are selected for movement within a turn. Rule select_ant, shown in Fig. 27 above, selects an ant randomly among the ones that have not yet moved this turn. The random linear exploration strategy requires that all matches be found, after which one is chosen randomly. This means that, in a single turn, the number of matches calculated for select_ant is quadratic in the number of ants (for instance, for 1,000 ants we have to calculate $\sum_{i=1}^{1,000} i = 500,500$ matches). Given that, in fact, the

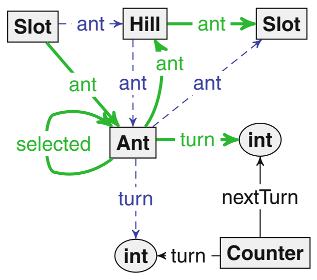


Fig. 30 Determinised ant selection rule

Table 3 Performance improvement due to rule system optimisation

Version	Time (s)	Reduction (%)
Start	510.7	
Determinised	245.8	50
Guided	124.0	51
Split	54.5	56

order in which ants move is hardly relevant to the behaviour of the system, we can improve the ant selection. A simple idea is to impose a linear order over the ants, reflecting their age (i.e., the order in which they are born), and letting ants move in this order. This means that (among other things) the rule in Fig. 27 changes to Fig. 30.

Simulation

We have simulated our rule system using the 64-bit Sun JVM 1.6 (build 16) with 1G (startup) to 4G (maximum) of memory, under Windows XP Pro x64, on a machine with 2 Intel Core CPUs at 3.00 GHz. The results are collected in Tables 3 and 4. All figures reported are averages over 5 runs. It should be noted, however, that (due to the random nature of the ant moves) the number of fields and ants, and hence the running time, show a large variance. This implies that not too much weight should be put on the exact figures. However, by looking at their relative values, we can see clearly the performance improvement.

Table 3 shows the running times for 100 turns using a succession of refined models:

Start This is the initial prototype, in which no performance tuning was done.

Determinised This is a version where the ant selection process was determinised using a fixed order of ants, as described above (see Fig. 30).

Guided This is a version in which the pattern matching strategy was aided manually with hints about the optimal search plan. The idea is to match “rare” edges first (see also [21]); this can be set through a GROOVE option.

Split This is a version where the move_random rule of Fig. 27 was replaced by a choice between four rules, each of which implements one of the choices in the regular expression.

It can be observed that especially the removal of nondeterminism in the ant selection had an enormous benefit on the performance. Although the table does not show this, we can also report that (as should be expected) the speedup factors grow for longer runs.

Table 4 shows the performance of GROOVE for simulation runs of increasing length. It is clear that the running times increase more than linearly with the size of the grid and the number of ants; moreover, the number of ants stays comfortably above the number of grid nodes. Given that the ratio of parts of food to grid nodes is 10:1; however, the number of ants is actually lower than might be expected.

In [25] and especially [20], an extensive complexity analysis of the AntWorld case can be found. An evaluation of the GROOVE performance with respect to that of other tools follows below.

5.4 Evaluation

Using GROOVE *for prototyping*

The example brings out the prototyping advantages of GROOVE very well. It is possible to encode the problem at hand directly, without having to think about or resort to special data structures. Graphs, rules and their effects can be inspected visually. Since GROOVE by default remembers entire transformation sequences, the debugging possibilities are good. The recent extension to type graphs also helps in maintaining consistency of the rule set.

Using control flow in GROOVE

GROOVE’s control language does quite well in capturing the intended system behaviour on a high level. Without the control language, it would have been necessary to include more auxiliary structures into the graphs themselves, in order to make sure that the phases as required in the problem description are indeed followed.

There is a possible extension to the control language that is quite interesting in the light of this case. The idea is to extend rule invocations with parameters. For instance, instead of using a selected-edge in the graph to distinguish an ant selected by select_ant, one could also use a variable in

Table 4 Results for increasing total turn counts

Turns	Time (s)	Rings	Fields	Ants
25	0.7	7	222	20
50	2.9	8	279	204
75	9.9	13	678	581
100	54.5	16	1,001	1,344
150	675.0	27	2,833	3,993

the control language to store the node identity. The function `move` would then become

```
function turn() {
        node x;
        while (select_ant(x)) {
                try { drop_food(x); }
                else try { pickup_food(x); }
                try { move_home(x); }
                else try { move_search(x); }
                else { move_random(x); }
        }
}
```

This both improves the understandability of the rule system and offers possibilities to improve the matching of the rules (see the performance section below).

Using GROOVE *for simulation*

This case does not involve full state space exploration (since there is a large amount of non-determinism in the ant moves, the state space size is truly enormous). Random linear exploration is used for simulation. This works well; the fact that (in GUI mode) GROOVE stores all intermediate steps and can display them, as well as the non-deterministic branches that were *not* taken, is a great help in constructing a correct solution.

Using GROOVE *for animation*

GROOVE essentially offers no facilities for animation. Simulation can be performed in automatic mode, but in that case the host graph is not animated.

The performance of GROOVE

One way to judge the performance is to compare our solution with other tools. In [15,20,25], we can find three other graph transformation-based solutions to this case, constructed in FUJABA, VMTS and VIATRA2, respectively. From the figures presented there, it is clear that those tools perform orders of magnitude faster than GROOVE; for instance FUJABA can simulate 1000 rounds in 17 seconds and VMTS does the same in 32 seconds. The only comparable solution is given by VIATRA2 with local matching only, which takes 800 seconds to simulate 150 rounds, compared to our average of 675 seconds. However, VIATRA2's incremental and hybrid matching algorithms improve upon this by orders of magnitude.

For this difference in performance, we offer the following explanations:

– The core functionality of GROOVE is to construct the state space, and this is what the performance is geared towards. In the AntWorld case, the only benefit we can draw from this is the ability to inspect traces that lead up to a certain result.
– GROOVE interprets all transformation rules, in contrast to FUJABA and VMTS which (partially) compile the rules to native code. In fact for FUJABA this is the core functionality: it is meant as a high-level modelling tool that produces Java code.
– Measurements have shown that around 90% of the GROOVE execution time is spent in matching. In contrast to VIATRA2 and VMTS, we have not yet invested much effort in optimising the search plans or implementing incremental algorithms (however, this is currently underway). A telling point is that, without these optimisations, indeed the performance of VIATRA2 is in the same order of magnitude as that of GROOVE— even though the reasons for the relatively poor performance, as analysed in [20], may not be the same.
  In FUJABA, the rules are in fact formulated in such a way that matching is trivial — which essentially means that writing "good" (fast) rules encompasses the manual creation of a good search plan.

To summarise…

The AntWorld simulation can be properly expressed in GROOVE. Key features of the tool relevant for the solution are: the random linear exploration strategy, attributed rules, and the control language. The performance of the tool and its lack of animation are an issue for this case study; however, the study also gives some hints on how to fine tune the performance of GROOVE after an initial solution for a problem is constructed.

6 Additional work

In this section, we present additional research that uses the GROOVE tool as part of the proposed solution for various problems. We only give a brief description of each work since they all utilise a subset of the GROOVE features that were presented in previous sections. The purpose here is to provide further examples that illustrate the usability of the tool in several different domains.

Control flow semantics of programming languages

The standard way to present the syntax of a programming language is by giving the syntax definition in (some variant of) Backus–Naur Form (BNF). On the other hand, there is no commonly accepted representation to describe a programming language semantics, which is usually only described in natural language. Any attempt at software verification suffers greatly from this, since natural language is inherently ambiguous and the semantics of common programming languages is usually fairly complex. In an attempt to solve this problem, Smelik, Rensink, and Kastenberg [36] propose to specify the control flow semantics of an imperative programming languages using GROOVE. In their work, they produce one or more graph transformation rules for each syntactic element of the language (expressions, conditionals, etc). Together, these rules not only formally describe the control flow semantics of the language but also can be used to construct the control flow graph (CFG) of programs. The input of this method is the Abstract Syntax Tree (AST) obtained from source code. In GROOVE, the AST is used as a start graph and its corresponding CFG is obtained by performing a linear exploration of the rules. The structure formed by AST+CFG (called a program graph) provides a complete static representation of the program, which can be used in program simulation or verification, for example. In order to show the feasibility of the proposed approach, the authors chose Java as their working language and developed rules in GROOVE to capture the control flow semantics of all language constructs, including exception handling.

Execution semantics of programming languages

In [22], Kastenberg, Kleppe, and Rensink describe the execution semantics of a simple object-oriented programming language in terms of graph transformation rules. A program graph (such as the one described in the previous item) is used as input and each rule application simulates the execution of a program instruction. By means of GROOVE's state space exploration capabilities, it is possible to generate finite execution traces of a program and model check for errors. In this setting, GROOVE can be seen as a non-deterministic execution engine for the language defined by the transformation rules. This correlates to other software model checking approaches such as the Java PathFinder project [40], on which the execution language is Java byte-code and the standard Java Virtual Machine is replaced by a non-deterministic one.

Computer-aided evolution of object-oriented designs

Evolution mechanisms are structures that prepare software for future changes. These mechanisms have to be implemented in the software from the get-go, which takes additional effort, but allows expected changes to be applied afterwards with minimum effort. In [6], Ciraci, van den Broek and Aksit introduce CDE, a tool that aids the application of evolution mechanisms using graph transformations expressed in GROOVE. The supported evolution mechanisms are expressed as (sequences of) fixed generic graph rules, which are instantiated by CDE with the relevant identifiers from the software to be changed. The software itself is expressed in ARGOUML [4] and is exported to a graph format using XMI. Then, the transformation is applied in GROOVE, and the output is imported again in ARGOUML. This entire process is carried out by CDE.

Aspect interference detection

Aspect Oriented Programming (AOP) is a paradigm of programming in which supporting functions are isolated from the main program's business logic. It aims to increase the modularity by allowing the separation of the cross-cutting concerns. An aspect can alter the behaviour of the base code by applying advice (additional behaviour) at various join points (points in the program). Aspects that in isolation behave correctly may interact when combined. A change made by interactions of aspects to each other's behaviour is called aspect interference. In [1], Aksit, Rensink and Staijen show an approach to detecting aspect interference. Aspect compositions are modelled in GROOVE as a graph production system. A graph-based model of a join point is generated from the source-code of the system. The run-time semantics of the AOP language is also specified as a graph transformation rule system. The graph-based model of the join point is transformed to a runtime-state representation. Combined with the production system, the execution of the aspects is simulated. The simulation results in an LTS, which is used for analysis and verification of the system at its join points.

Semantics of activity diagrams

There is much research done about formal modelling and verification of workflows using different formal languages.

[4] http://argouml.tigris.org.

In [13] and [18], Engels and Hausmann have introduced the notion of dynamic metamodelling (DMM) as a semantics description technique for Visual Modelling Languages. Graph transformation is used to define the behaviour as a system of transitions. The traditional graph rules were extended in their work by defining a new concept of *rule invocation*. There are two kinds of rules in DMM: *big-step* and *small-step* rules. Big-step rules act as traditional rules and small-step rules should be invoked by big-step rules. Using these kinds of rules, modelling of complex systems can be simplified. Hausmann then defines semantics for UML activity diagrams using the concept of DMM. Subsequently, Soltenborn [37] uses DMM and defines semantics for UML activity diagrams for modelling and verification of workflows. He uses GROOVE to perform such verification.

Modelling Dynamic Reconfigurations

In [24], Krause et al. propose an approach for defining reconfigurations for the coordination language Reo [2] using graph-rewriting techniques. In their work, they apply the ideas of high-level-replacement (HLR) systems to the coordination language Reo and show how they can be used to model dynamic reconfigurations of Reo connectors. They also provide a full implementation of this reconfiguration approach for Reo, including tools for defining, verifying and executing dynamic reconfigurations. For verification, they have implemented conversion tools that produce output for the Attributed Graph Grammar (AGG) system [38] and GROOVE. Using these two tools, Krause et al. perform confluence and termination checks for reconfiguration rules as well as state space exploration and model checking of dynamic reconfigurations.

Applying formal methods to gossipping networks

A gossipping network consists of a large number of nodes that communicate with adjacent nodes only, spreading information in the same way people spread gossip through a community. In [7], Crouzen, van de Pol and Rensink apply formal methods to analyse properties of gossipping networks. The applied methods and tools include μCRL2, GROOVE, Continuous Time Markov Chains, Markov Reward Models and model checking. In this whole, GROOVE is used for describing the behaviour of the gossipping network, and for applying symmetry reduction to detect and remove equivalent states. This allows bigger networks to be handled by the formal methods. The symmetry reduction is realised in GROOVE by the isomorphism check that is applied automatically when exploring a state space.

7 Conclusion

To conclude, we first give an overview of the case studies presented in this paper. Following the overview, we discuss future extensions and features planned for GROOVE. Last, we give a short comparison between GROOVE and other graph transformation tools and make some final remarks.

7.1 Overview of case studies

The case studies discussed in Sects. 2–5 have quite different characteristics and the GROOVE solutions presented stress different features of the tool. Table 5 shows an overview of the main points discussed in each section.

The first line of the table shows the general area of each case. It is reasonable to assume that problems from similar areas may have a similar solution in GROOVE. The following three lines present design choices that the user must make when modelling a problem in GROOVE:

Typing The use of type graphs considerably changes the degree of freedom in modelling. Untyped graphs do not impose any restrictions and allow the fast conception of an initial solution. Typed models may take longer to develop, but ensure a certain consistency in the solution and ease the presentation. It is evident that certain kind of problems may benefit from typing, e.g., the BPMN to BPEL case, whereas problems with few typing structure, such as in the leader election case, have little to gain in a typed setting. As a rule of thumb, at least node types should be used. They impose virtually no modelling overhead and improve readability of the solution.

Control All cases presented use a method to control rule application. The control method may change during the modelling process. An initial design usually starts with no control and then moves to rule priorities when necessary. If the interaction between rules becomes complex, for example when a set of rules may disable/enable several other rules at different priority levels, then control programs are normally used.

Strategy The strategy used for state space exploration has a large impact on the performance of the tool. The strategy choice depends on the characteristics of the problem. Cases where the order of rule application is irrelevant and rule application always leads to a single final state (confluent grammar) usually employ a linear exploration strategy, e.g., the BPMN to BPEL case. It is the opposite for cases, where the interleaving of rule applications is crucial. Full state space exploration is usually necessary for the verification of dynamic behaviour (leader election case). Finally, when the state space is too large, partial exploration may be used for bug hunting (security analysis case).

Table 5 Overview of the case studies presented

Case	BPMN to BPEL	Leader election	Organisational security	AntWorld
Area	Model transformation	Verification	Analysis	Simulation
Typing	No, but would be useful	No, and would not be useful	Node types	Full types
Control	Rule priorities	Rule priorities	Control program	Control program
Strategy	Linear exploration	Full state space exploration	Find rule application	Random linear exploration
Relevant features	Quantified rules Wildcards and regular expressions	Model checking Symmetry reduction by isomorphism checking	Quantified rules	Attributes
Interface	GUI helps debugging	GUI helps prototyping	GUI helps analysis of results	GUI helps prototyping
Strong points	Rapid prototyping Local confluence check	Analysis capabilities	Rule expressiveness Analysis capabilities	Rapid prototyping
Weak points	Model trafo support Interoperability	Scalability	Scalability	Performance Animation

The fifth line of Table 5 summarises which features of GROOVE were particularly relevant for the solution of the case study. These features have been discussed in depth in the corresponding sections.

A final solution is often reached after some refinement iterations (as illustrated in the AntWorld case), and each iteration gives new insights on the problem being handled and provides an idea on where to improve next. In all cases the interactivity of GROOVE's graphical interface was very useful and helped in the solution development cycle.

7.2 Future extensions

Based on the case studies carried out, we are working on and planning some tool extensions that will further enhance the usability and power of GROOVE.

Performance improvements. A key factor in most case studies is the performance of GROOVE. We are investigating two ways to improve performance. First, using incremental pattern matching, as studied in [4], we expect a big performance increase over the current matching algorithm. Second, abstraction, as studied in [32], is expected to result in smaller overall state spaces, which will be particularly advantageous for the verification-type case studies, such as the leader election and security cases.

Control parameters. We are working on an extension to the control language with *rule parameters*. These parameters will allow a more fine-grained control over the place in a graph where a rule should be applied. For instance, if a sequence of rules should all be applied to the same node, currently the first rule has to mark that node with a special edge and the subsequent rules have to test for that edge. Using parameters, the control program would specify directly that the subsequent rules have to match at the node "found" by the first rule. (Note that the control languages of several of the tools discussed below, including at leastGRGEN, VMTS and VIATRA2, already support parameter passing.)

Transactions. A single rule expresses an atomic change to a graph, but not all atomic changes can be captured by single rules. Quantification extends the expressiveness of rules enormously, but there are still many cases in which one would like to specify an atomic change that is too complex to be expressed by a single rule. We therefore intend to implement a notion of graph transaction, which atomically combines a (controlled) set or sequence of rule applications. This will also help in state space reduction, since such transactions cannot be "interrupted" by other rule applications.

7.3 Comparison with other tools

In this section, we compare GROOVE with other general graph transformation tools and tools which use graph transformation as an engine to achieve some other goal, like model transformation.

The comparison is summarised in Table 6 and covers seven different criteria. These criteria have been chosen based on the key features used in solving the cases presented in this article (see Table 5).

The first criterion is the *focus* of the tools, i.e., the main goal for which these tools are designed and optimised. We have four different categories: general purpose, model transformation, high performance, and verification. We call a tool a high performance tool if it incorporates design decisions that increase performance, possibly at the cost of generality; for instance by restricting the allowed graph structures to reflect programmable data structures. It should be noted that the focus criterion only shows the emphasis of the tools and does not imply restrictions on usage. For instance, GROOVE is categorised as a general purpose tool, however, in Sect. 2 we saw that GROOVE can be used for model transformation as well. Similar remarks hold for other tools.

The second criterion is *typing*. All tools except AUGUR support type graphs or meta models and in all tools except GROOVE, type graphs are mandatory, i.e., models must have a type graph. GROOVE is the only tool in which the use of typing is optional.

The third criterion is the *control* functionality of the tools. In most tools the order of rule applications can be controlled using an imperative language. In VIATRA2, FUJABA, VMTS, and PROGRES, this language has advanced features like recursion. AGG and ATOM3 only support priorities. The control language of GROOVE is not as advanced as some of the other tools, but GROOVE supports priorities for rules, as well. Additionally, GROOVE has advanced quantification features which are compared separately in another criterion.

The fourth criterion is the ability of the tools with respect to rule application (*state space exploration*) strategies. Most of the tools support more than one strategy, such as random, manual, rule priority-based, or customised through the use of the control language. However, all these variant strategies only explore one linear trace of rule applications. GROOVE, however, can explore the entire state space generated from different rule application sequences. In fact, it supports multiple full state space exploration strategies. AUGUR also provides full state space exploration.

The fifth criterion concerns advanced rule features, that increase the expressiveness of individual rules. As seen throughout this paper, GROOVE supports nested quantification. In this respect, no other tool is capable of specifying graph conditions as concise as GROOVE [33]. GRGEN, PROGRES, VIATRA2, and VMTS support one level of universal quantifier, which are rules that are entirely universally quantified. It means that first all the matches of one rule are found then the rule is applied on all the matches concurrently. PROGRES and FUJABA have *set nodes* which are in fact single universal quantified nodes. GREAT provides *match conditions* which is a limited version of universally quantified rules. VIATRA2 supports recursive rules in the pattern definition, besides patterns can be defined independent of rules and consequently, can be reused in the definitions of other patterns or rules. GROOVE supports wildcards on edge labels, allowing paths in graphs to be specified using regular expressions. PROGRES, FUJABA and GRGEN also support regular expressions. The graph grammars supported by AUGUR are a bit restricted as it for example does not support node deletions and node merging.

The sixth criterion is about analysis facilities that are provided by different tools. AGG has one particular feature, namely, critical pair analysis of rules, which checks whether

Table 6 Comparison between GROOVE and other tools

Tool	Focus	Typing	Control	Exploration	Advanced rule features	Analysis	Editing
AGG [38]	General purpose	Required	Priority	Linear		Critical pairs	Graphical
ATOM3 [8]	Model transformation	Required	Priority	Linear	Triple Graph Grammar rules		Graphical
AUGUR [23]	Verification	Untyped		Exhaustive		Abstraction	Textual
FUJABA [14]	High performance	Required	Imperative (advanced)	Linear	Set nodes Regular expressions		Graphical
GREAT [3]	Model transformation	Required	Imperative	Linear	Match condition Recursive patterns		Graphical
GRGEN [16]	High performance	Required	Imperative	Linear	Regular expressions Universal quantification		Textual
PROGRES [35]	General purpose	Required	Imperative (advanced)	Linear	Set nodes Star rules Regular expressions		Graphical
VIATRA2 [39]	Model transformation	Required	Imperative (advanced)	Linear	Recursive patterns Universal quantification	Constraint satisfaction problems	Textual
VMTS [41]	Model transformation	Required	Imperative (advanced)	Linear	Universal quantification		Textual
GROOVE	General purpose	Optional	Imperative Priority	Multiple	Quantification Wildcards Regular expressions	Model checking	Graphical

two rules interfere with each other. This feature is used to determine statically if a graph grammar is confluent. AUGUR generates a Petri net based on the graph grammar whose state space is an over approximation of the graph grammar. Using this abstraction technique it can analyse graph grammars with infinite state spaces. VIATRA2 provides some limited support for solving constraint satisfaction problem on graph models and finally, GROOVE can verify CTL and LTL specified properties on the state spaces generated by a graph grammar.

The final criterion is whether a tool provides a graphical user interface for editing graphs and rules, or is text-based only.

7.4 Final remarks

In this paper we give a flavour of how systems can be modelled and analysed with our graph transformation tool, GROOVE. The case studies presented cover quite different domains, which, together with the additional work given in Sect. 6, demonstrates that GROOVE is a flexible tool that can be used to solve problems from several different areas.

Another important point is that GROOVE is very easy to install and use. The interactive GUI helps the user to experiment with, analyse, and improve the grammar constructed. This implies that GROOVE is eminently suited for fast prototyping.

The grammars for the case studies discussed in this paper are available at the GROOVE project website (http://groove.cs.utwente.nl/downloads/). The binaries and source code of the tool can also be downloaded from the same address, as well as some documentation, such as a user manual and tutorials.

Acknowledgments This study was supported by CHARTER (EU/ARTEMIS 100039). E. Zambon was supported by GRAIL (NWO 612.000.632). M. Zimakova was supported by GRASLAND (NWO 612.063.408).

References

1. Aksit, M., Rensink, A., Staijen, T.: A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In: AOSD '09, Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development, pp 39–50. ACM, New York, NY, USA (2009)
2. Arbab, F.: Reo: A channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(03), 329–366 (2004)
3. Balasubramanian, D., Narayanan, A., van Buskirk, C., Karsai, G.: The graph rewriting and transformation language: great. In: Proceedings of the Third International Workshop on Graph Based Tools (GraBaTs 2006). ECEASST., vol. 1. EASST (2006)
4. Bergmann, G., Horváth, Á., Ráth, I., Varró, D.: A benchmark evaluation of incremental pattern matching in graph transformation. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G., (eds.) International Conference on Graph Transformations (ICGT). LNCS., vol. 5214. pp 396–410. Springer, Berlin (2008)
5. Chang, E., Roberts, R.: An improved algorithm for decentralized extrema-finding in circular configurations of processes. Comm. ACM **22**(5), 281–283 (1979)
6. Ciraci, S., van den Broek, P., Aksit, M.: Framework for computer-aided evolution of object-oriented designs. In: 32nd Annual IEEE International Computer Software and Applications. COMPSAC '08, pp 757–764 (2008)
7. Crouzen, P., van de Pol, J.C., Rensink, A.: Applying formal methods to gossiping networks with mCRL and GROOVE . ACM SIGMETRICS Perform. Eval. Rev. **36**(3), 7–16 (2008)
8. de Lara, J., Vangheluwe, H.: ATOM3 : a tool for multi-formalism and meta-modeling. In: Fundamental Approaches to Software Engineering (FASE). LNCS., vol. 2306, pp 174–188 (2002)
9. de Mol, M.J., Zimakova, M.V.: A GROOVE solution for the BPMN to BPEL model transformation. Technical Report TR-CTIT-09-31, Centre for Telematics and Information Technology, University of Twente, Enschede (2009)
10. Dikmans, L.: Transforming BPMN into BPEL: Why and How. Oracle Technology Network (2008). http://www.oracle.com/technology/pub/articles/dikmans-bpm.html
11. Dimkov, T.: Portunes security framework. http://sourceforge.net/projects/portunes/
12. Dimkov, T., Pieters, W. P. H.: Portunes: representing attack scenarios spanning through the physical, digital and social domain. In: ARSPA-WITS, Springer, Berlin (2010)
13. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic Meta-Modeling: a graphical approach to the operational semantics of behavioral diagrams in UML. In: Evans, A. Kent, S. B.S., ed.: Proceedings of the 3rd international conference on the Unified Modeling Language (UML 2000), York (UK), pp 323–337. Springer, Berlin (2000)
14. The FUJABA Toolsuite. (2006). http://www.fujaba.de
15. Geiger, L., Zündorf, A.: FUJABA case studies for GraBaTs 2008—lessons learned. Software Tools for Technology Transfer. STTT **12**(3–4), pp 287–304 (2010)
16. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: GRGEN: a fast SPO-based graph rewriting tool. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds) International Conference on Graph Transformations (ICGT). LNCS, vol. 4178, pp 383–397. Springer, Berlin (2006)
17. 5th International Workshop on Graph-Based Tools (the contest). (2009).http://is.ieis.tue.nl/staff/pvgorp/events/grabats2009/.
18. Hausmann, J.H.: Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages. PhD thesis, University of Paderborn, Germany (2005)
19. Heckel, R.: Graph transformation in a nutshell. Electr. Notes Theor. Comput. Sci. **148**(1), 187–198 (2006)
20. Horváth, Á., Bergmann, G., Ráth, I., Varró, D.: Experimental assessment of combining pattern matching strategies with VIATRA2. Software Tools for Technology Transfer. STTT **12**(3–4), pp 211–230 (2010)
21. Horváth, Á., Varró, G., Varró, D.: Generic search plans for matching advanced graph patterns. In: Ehrig, K., Giese, H. (eds) Graph Transformation and Visual Modelling Techniques (GT-VMT). Electronic Communications of the EASST, vol. 6 (2007)
22. Kastenberg, H., Kleppe, A., Rensink, A.: Defining object-oriented execution semantics using graph transformations. In: Gorrieri R., Wehrheim H. (eds.) Formal Methods for Open Object-Based Distributed Systems (FMOODS). LNCS, vol. 4037, pp. 186–201. Springer, Berlin (2006)
23. König, B., Kozioura, V. (2008) AUGUR—a new version of a tool for the analysis of graph transformation systems. ENTCS 211 201–210

24. Krause, C., Maraikar, Z., Lazovik, A., Arbab, F.: Modeling dynamic reconfigurations in Reo using high-level replacement systems. Sci. Comput. Program. **76**(1), 23–36 (2011)
25. Mészáros, T., Mezei, G., Levendovszky, T., Asztalos, M.: Manual and automated performance optimization of model transformation systems. Software Tools for Technology Transfer. STTT **12**(3–4), pp 231–243 (2010)
26. Object Management Group: Business Process Model and Notation, V1.2 (2009). http://www.omg.org/spec/BPMN/1.2/
27. Organization for the Advancement of Structured Information Standards: Web Services Business Process Execution Language, V2.0 (2007). http://docs.oasis-open.org/wsbpel/2.0/wsbpel-2.0.pdf
28. Ouyang, C., Dumas, M., ter Hofstede, A., van der Aalst, W.: Pattern-based translation of BPMN process models to BPEL web services. Int. J Web Serv. Res. (JWSR) **5**(1), (2008)
29. Ouyang, C., van der Aalst, W., Dumas, M., ter Hofstede, A.: Translating BPMN to BPEL. Quensland University of Technology, Brisbase, Australia. E-Print, revised version (2006). http://eprints.qut.edu.au/5266/
30. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B., (eds.) Applications of Graph Transformations with Industrial Relevance, (AGTIVE). LNCS, vol. 3062, pp. 479–485, Springer, Berlin (2004). http://sourceforge.net/projects/groove.
31. Rensink, A.: Representing first-order logic using graphs. In: Ehrig, H.,Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) International Conference on Graph Transformations (ICGT). LNCS, vol. 3256, pp. 319–335. Springer, Berlin (2004)
32. Rensink, A., Distefano, D.S.: Abstract graph transformation. In: Mukhopadhyay, S., Roychoudhury, A., Yang, Z. (eds.) Software Verification and Validation, Manchester. Electronic Notes in Theoretical Computer Science, vol. 157, pp. 39–59. Elsevier (2006)
33. Rensink, A., Kuperus, J.H.: Repotting the geraniums: on nested graph transformation rules. In: Boronat, A., Heckel, R. (eds.) Graph transformation and visual modelling techniques (GT-VMT). Electronic Communications of the EASST., EASST, vol. 18 (2009)
34. Rensink, A., Van Gorp, P.: Graph transformation tool contest 2008. Software Tools for Technology Transfer. STTT **12**(3–4), pp 171–181 (2010)
35. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES approach: language and environment. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) Handbook of graph grammars and computing by graph transformation: applications, languages, and tools. vol. 2, pp. 487–550. World Scientific Publishing Co., Inc., (1999)
36. Smelik, R., Rensink, A., Kastenberg, H.: Specification and construction of control flow semantics. In: Grundy, J., Howse, J., eds.: Visual Languages and Human-Centric Computing (VL/HCC), pp. 65–72. IEEE Computer Society Press, Brighton, UK, Los Alamitos (2006)
37. Soltenborn, C.: Analysis of UML Workflow diagrams with dynamic Meta Modeling Techniques. Master's thesis, University of Paderborn, Germany (2006)
38. Taentzer, G.: AGG: A graph transformation environment for modeling and validation of software. In: Applications of Graph Transformations with Industrial Relevance, (AGTIVE). LNCS, vol. 3062, Springer (2004) 446–453
39. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Progr. **68**(3), 187–207 (2007). http://www.eclipse.org/gmt/VIATRA2/.
40. Visser, W., Havelund, K., Brat, G.P., Park, S.: Model checking programs. Autom. Softw. Eng. **10**(2); 203–232 (2003)
41. Visual Modeling and Transformation System (2008). http://www.aut.bme.hu/Portal/Vmts.aspx?lang=en.
42. W3C: XSL Transformations (XSLT), V1.0, Recommendation. (1999). See http://www.w3.org/TR/xslt.